8.1 INTRODUCTION

Suppose a person want to transfer Rs.1,000 from an account A having balance Rs.10,000 to an account B which have a balance of Rs.15,000. There are various operations involved in this like, checking the balance in account A, deducting Rs.1,000 from account A, adding Rs.1,000 to account B etc. But according to a customer view point all these operations forms a single operation. Therefore transaction can be defined as collection of various operations related to database processing that forms a single logical unit of work. Transaction finds their application in railways, airlines, banking system, etc.

When a transaction begins, all operations involved in it must be done completely or not at all. External world see only the beginning or end of a transaction. If the transaction execute successfully, then database is updated otherwise all operations must be rolled back. Following figure shows the structure of a transaction.

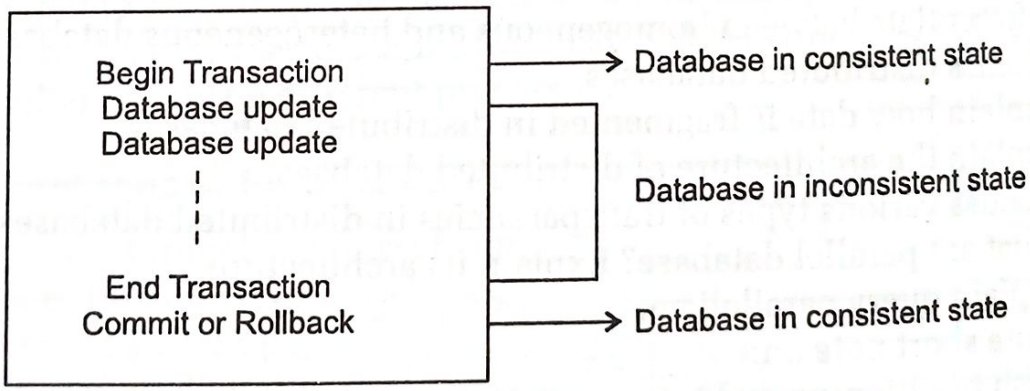


Fig. 8.1: Structure of a Transaction

8.2 TRANSACTION PROPERTIES

(GGSIPU 2009, 2011; MDU, Dec. 2009, May 2010; PTU 2012, 2011)

A transaction should follow the following ACID properties

1. **Atomicity (A):** A transaction should be atomic means that either all operations of transaction complete successfully or none of them is implemented.

2. **Consistency (C):** A database must be in consistent state before the start of transaction and at the end of the transaction. A transaction must be executed in isolation with no other transaction executing concurrently. This means that the data used by transaction T1 cannot be changed by the transaction T2 at same time.
3. **Isolation (I):** When multiple transactions are executing simultaneously then no two transactions should interfere with each other. For every pair of transaction T1 and T2, it appears to T2 that either T1 finished first before T2 started or T1 started execution after T2 finishes.
4. **Durability (D):** After successful completion of a transaction, the changes made to database should be permanent even if there are any failures.

Now we shall explain the ACID properties on the following transaction T1.

T1
read (A)
A = A-1000
write (A)
read (B)
B = B+1000
write (B)

Fig. 8.2: Transaction T1

Before transaction starts, suppose A = Rs.5,000 and B = Rs.8,000.
So, after transaction ends the value of
A = Rs.4,000 and B = Rs.9,000.

Atomicity

If T1 fails after deducting Rs. 1,000 from A and before crediting it to B, the system would be in inconsistent state, as account A would have Rs. 4,000 and account B would have Rs. 8,000. This means Rs. 1,000 has been lost from account B.So, system should ensure that updates of partially executed transaction are not reflected to database.

Consistency

Transaction T1 before executing its first statement sees the database in consistent state, i.e. sum of balances in account A and account B as Rs. 13,000. Suppose T1 fails after deducting Rs.1,000 from account A and before crediting it to account B. Then after T1 ends, database would be in inconsistent state as it would see the sum of balances of account A and B as Rs. 12,000 instead of Rs. 13,000. So, updates done by T1 should not be reflected.

Isolation

Suppose while transaction T1 is executing its statement an another transaction T2 is allowed to access the database. This is shown in following example:

T_1	T_2
read (A)	
A=A-1000	
write (A)	
	read (B)
	B=B*2
	write (B)
read (B)	
B = B +1000	
write (B)	

Fig. 8.3: Example of a Transaction

In the above example transaction T2 will see an inconsistent database. It sees an inconsistent value of B i.e. Rs. 8,000 and will updates it to 16,000. But actually T2 should have waited for T1 to commit and then read the value of B as 9,000 and updated it to 18,000. So, to ensure isolation these transactions should be executed serially.

Durability

When transaction T1 is fully completed i.e. T1 commit then, the updates done by T1 should persist (i.e. A = Rs. 4,000, B = Rs. 9,000 even if there is any failure).

8.3 ISOLATION LEVELS

There are various level of isolation that provides a measure of influence of concurrent transaction on a given transaction. Various isolation levels are:

(i) **Serializable:** A transaction does not overwrite data updated by another process of other transaction i.e. all transactions are executed serially one after another.

(ii) **Repeatable reads:** All the read and write locks are kept till end of the transaction.

(iii) **Read committed.** A write lock is kept until the end of transaction but a read lock is released as soon as data is read. Non-repeatable read phenomenon can occur.

(iv) **Read uncommitted.** One transaction can read not yet committed updates done by some other transactions i.e. dirty reads are allowed.

8.4 TRANSACTION STATES

(GGSIPU 2009-2010; MDU, Dec. 2009, May 2010)

A transaction during its entire lifetime is in one of the following states:

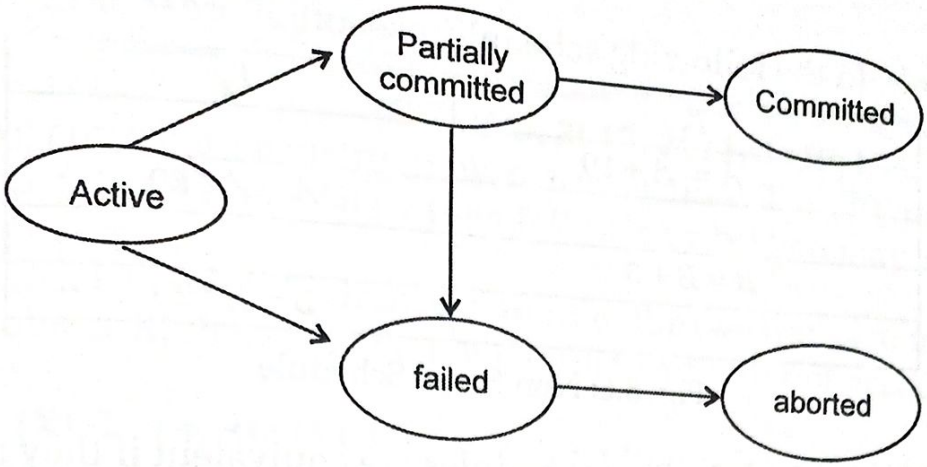


Fig. 8.4: Transaction States

1. **Active State:** A transaction is in this stage while it is executing.
2. **Partially Committed:** When final statement has been executed.
3. **Failed:** When normal execution can no longer proceed.
4. **Aborted:** After transaction has been rollbacked and database has been restored to the state prior to start of transaction.
5. **Committed:** When transaction has completed successfully.

8.5 SCHEDULES

Transaction processing system allows multiple transaction to execute concurrently. All these transactions should be executed in some schedules so that consistency of database can be maintained.

A schedule refers to execution of a set of transaction. Various types of schedules are:

1. **Serial schedule:** Various transactions are executed strictly according to a sequence, one at a time.
   Example: In the following schedule first T1 is executed completely i.e. each Instruction of T1 is executed completely, then T3 is executed and at last T2.

T_1	T_2	T_3
$A = A - 5$ $B = B + 25$		
		$A = A - 50$ $B = B / 20$
	$A = A + 20$ $B = B - 5$	

Fig. 8.5: Serial Schedule

2. **Non-serial schedule:** In this type of schedule operations of different transactions are interleaved.

Example: In the following schedule, execution order is T1→T2→T1→T2.

T_1	T_2
$A = A+10$	
	$A = A*50$
$B = B+6$	
	$B = B*20$

Fig. 8.6: Non-Serial Schedule

3. **Equivalent schedule:** Two schedules are equivalent if they produce same result on an initial database state.
   Example: Following two schedules are equivalent:

T_1	T_2
$A = A+10$	
	$B = B+10$
$A = A+50$	

T_1	T_2
	$B = B+10$
$A = A+10$	
$A = A+50$	

Fig. 8.7: Equivalent Schedule

4. **Serializable Schedule:** If the result of execution of a set of operation of a non-serial schedule is equivalent to some serial execution of transaction then the schedule is a serializable schedule.

(UPTU 2011-12; PTU, 2012]

Example: The following schedule T1 → T2 → T1 → T2

T_1	T_2
read (A)	
	read (B) read (A)
write(A)	
	write(B)

is a serializable schedule, if we execute T2 first and then T1 as shown below.

T_1	T_2
	read (B) read (A) write (B)
read (A) write (A)	

Fig. 8.8: Serialization Schedule

8.6 CONFLICT SERIALIZABILITY

(GGSIPU, 2013, 2011, 2010; UPTU 2011-12)

Let I_i and I_j be two consecutive instructions of transactions T_i and T_j respectively. If both I_i and I_j operate on same data x then following situation may arise.

1. **I_i = read (X), I_j = read (X).** If instructions of the two transactions just read the data object X, then they do not conflict and order of execution is not important.
2. **I_i = read (X), I_j = write (X).** Order of execution is important because one instruction reads data, modified by the other instruction.
3. **I_i = write (X), I_j = read (X).** Order is important because one instruction reads data modified by the other instruction.
4. **I_i = write (X), I_j = write (X).** Order of execution is important because the next read (X) operation will read the value of the latter of the two write instructions.

We conclude that two instructions of two different transactions operating on same data conflicts when one of them is a write operation.

If one schedule can be converted to other schedule by a series of swaps of non-conflicting instructions, then the two schedules are said to be conflict equivalent.

A schedule is said to be conflict serializable if it is conflict equivalent to a serial schedule.

Example 1: Consider the following schedule. Determine if the schedule is conflict serializable? Also find a serial schedule.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

(Conflict cannot swap)

(No conflict ∵ operation on different data object, so swap)

T_1	T_2
read (A)	
write (A)	
read (B)	
	read (A)
	write (A)
write (B)	
	read (B)
	write (B)

(No conflict so swap)

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Fig. 8.9: Schedule

Schedule is conflict serializable and serial schedule is T1 → T2

Example 2: Let T1, T2, T3 be three concurrent transaction. Determine following is conflict serialization or not. For a serialisable schedule, determine the equivalent serial schedule. **(GGSIPU, 2013, 2009)**

(a) r1 (x); r3 (x); w1 (x); r2 (x); w3 (x)

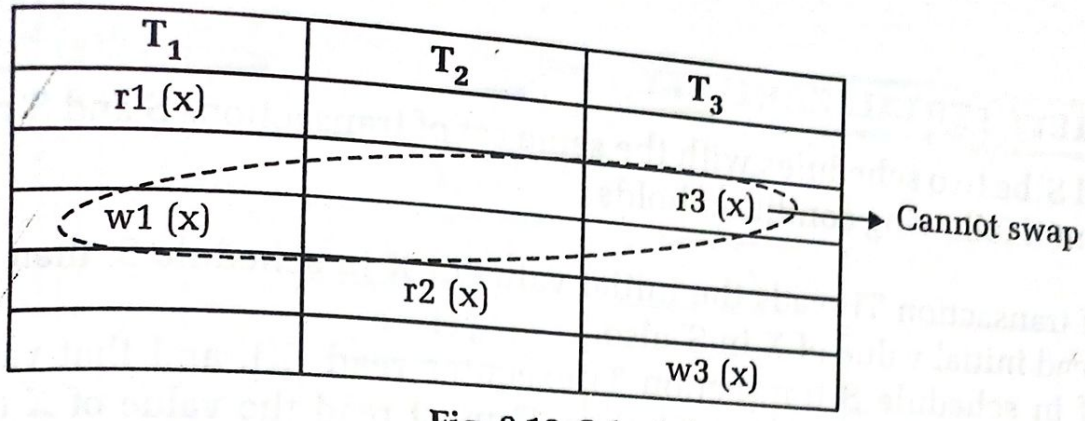


Fig. 8.10: Schedule

Schedule is non-serializable because T3 reads data object X before T1 write X. So, cannot be swapped.

(b) r1 (x); r3 (x); w3 (x); w1 (x); r2 (x) **(GGSIPU, 2013)**

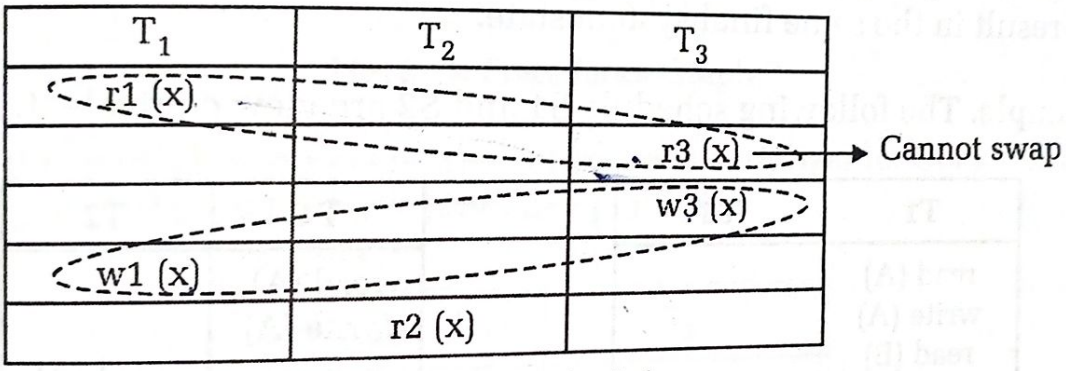


Fig. 8.11: Schedule

Schedule is not serializable.

(c) r3 (x); r2 (x); w3 (x); r1 (x); w1 (x) **(GGSIPU, 2013)**

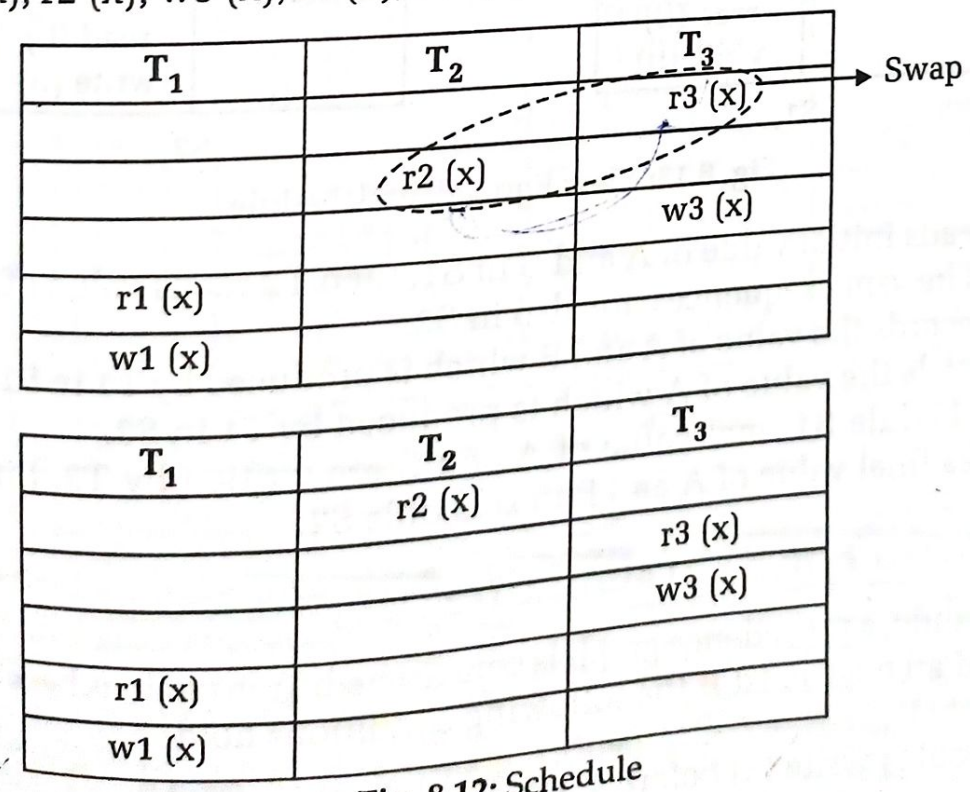


Fig. 8.12: Schedule

Schedule is serializable.

Serial schedule is T2 → T3 → T1

8.7 VIEW SERIALIZABILITY

Let S and S' be two schedules with the same set of transactions S and S' are view equivalent if following condition holds

1. If transaction Ti reads the initial value of X in schedule S, then Ti must read initial value of X in S' also.
2. If in schedule S transaction Ti executes read (X), and that value was produced by transaction Tj, then Ti must read the value of X that was proceduced by the same write (X) operation of Tj.
3. The transaction (if any) that performs the final write (X) operation in schedule S must also perform the final write (X) operation in schedule S' Condition (1) and (2) ensures that each transaction reads the same value in condition (3) alongwith condition (1) and (2) ensures that both schedules result in the same final system state.

Example. The following schedule S1 and S2 are view equivalent.

T1	T2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

S1

T1	T2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

S2

Fig. 8.13: View Equivalent Schedule

- T1 reads initial value of A and B in S1, then T2 reads value of A and B in S1. The same sequences are also in S2.
- T2 records the value of A and B which is produced by T1 in S1. Similarly, T2 reads the value of A which is produced by T1 in S2.
- In schedule S1 final value of A and B are written by T2. Similarly, T2 writes final value of A and B in schedule S2.

8.8 TESTING FOR SERIALIZABILITY

For any schedule, a precedence graph is constructed. Transactions become vertices of graph and an edge exist if the following conditions hold.

1. T_i executes write (x) before T_j executes read (x)
2. T_i executes read (x) before T_j executes write (x)
3. T_i executes write (x) before T_j executes write (x)

Consider following schedule

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)

Precedence graph of above schedule is

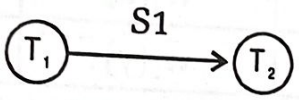


Fig. 8.14: Precedence Graph

Since the graph has no cycle, the schedule is serializable. Consider another schedule given below with its precedence graph :

T_1	T_2
read (A) write (A)	
	read (A) write (A) read (A)
write (B) read (B) write (B)	
	write (B)

Fig. 8.15: Precedence Graph

Since, the graph has cycle so, the schedule is non-serialisable.

Example: Draw precedence graph and check whether the schedule is conflict serializable or not.

T_1	T_2	T_3
$r_1(x)$		
	$r_2(z)$	
$r_1(z)$		
		$r_3(x)$
		$r_3(y)$
		$w_3(x)$
	$r_2(y)$	
	$w_2(z)$	
	$w_2(y)$	

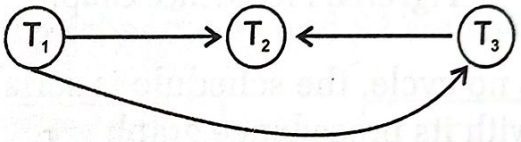


Fig. 8.16: Schedule with its precedence graph

Since, the precedence graph contains no cycle. Hence it is conflit serializable.

8.9 RECOVERABILITY

Any transaction is subjected to failure due to several reasons. Due to atomicity property, all operation of a transaction should be executed successfully or none of them. So whenever a transaction T_i fails all its effect needs to be undone so that database comes in consistent state. Moreover if some other transaction T_j uses the result of T_i, then T_j also needs to be aborted.

8.9.1 Recoverable Schedule

T1	T2
read (A)	
write (A)	
	read (A)
read (B)	

Fig. 8.17 : Schedule

In above schedule assume transaction T2 commits after reading the data value A which was written by transaction T1. Now if T1 fails, T2 should be aborted since T2 reads a data value modified by T1. But T2 cannot be aborted since it has already been committed. Therefore there is no way to recover from this situation and hence database is in inconsistent state.

A schedule is said to be recoverable if for each pair of transaction T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before commit operation of T_j.

8.9.2 Cascadeless Schedules

To recover from a failure of transaction T_i correctly we may have to rollback several transactions that are directly or indirectly dependent on T_i. This series of transaction rollback is known as cascading rollback.

T1	T2	T3
read(A)		
read(B)		
write(B)		
	read(B)	
	write(B)	
		read(B)
		write(B)

Fig. 8.18: Schedule

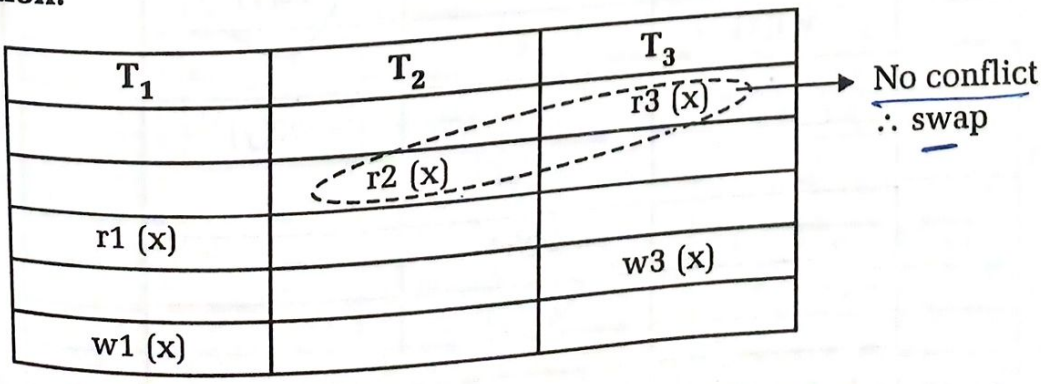
In above schedule transaction T1 updates a data value B which is read and updated by T2. Further transaction T3 reads this value of B and updates it. Now if T1 fails, T1 must be rolled back. Since T2 is dependent on T1, T2 should be rolled back. Moreover T3 is also dependent on T2 so T3 should also be rolled back.

SOLVED EXAMPLE

Example 1. Determine whether the following schedule is conflict serializable? **(GGSIPU, 2013, 2009)**

r3 (x); r_2 (x); r_1 (x); w_3 (x); w_1 (x)

Solution.



T_1	T_2	T_3
	r2 (x)	
		r3 (x)
r1 (x)		
		w3 (x)
w1 (x)		

Cannot swap

Fig. 8.19: Schedule

Since, T3 reads X before T1 reads X but T1 reads X before T3 write X. So, schedule is not serialisable.

Example 2. Consider three transaction T1, T2, T3 and schedule S1 and S2. Draw serializability graphs for S1 and S2 and state whether each schedule is serializable or not. If a schedule is serializable, write down equivalent serial schedule.

T1: r1 (x); r1 (z); w1 (x)
T2: r2 (z); r2 (y); w2 (z); w2 (y)
T3: r3 (x); r3 (y); w3 (y)
S1: r1 (x); r2 (z); r1 (z); r3 (x); r3 (y); w1 (x); w3 (y); r2 (y); w2 (z) ; w2 (y)
S2: r1 (x); r2 (z); r3 (x); r1 (z); r2 (y); r3 (y) ; w1 (x); w2 (z); w3 (y) ; w2 (y)

Solution. Schedule S1

T_1	T_2	T_3
r1 (x)		
	r2(z)	
r1(z)		
		r3(x)
		r3(y)
w1(x)		
		w3(y)
	r2(y)	
	w2(z)	
	w2(y)	

T3 reads x and y before T1 modifies X and T2 reads/modifies y. So we can swap.

T_1	T_2	T_3
		r3(x)
		r3(y)
r1(x)		w3(y)
	r2(z)	
r1(z)		
w1(x)		
	r2(y)	
	w2(z)	
	w2(y)	

T_1	T_2	T_3
		r3(x) r3(y) w3(y)
r1(x) r2(z) w1(x)		
	r2(z) r2(y) w2(z) w2(y)	

Fig. 8.20: Schedule

Hence S1 is serializable schedule and serializability graph is

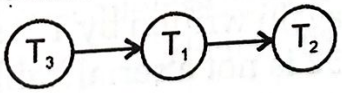


Fig. 8.21: Precedence Graph

Schedule 2

T_1	T_2	T_3
r1(x)		
	r2(z)	r3(x)
r1(z)		
	r2(y)	
		r3(y)
w1(x)		
	w2(z)	
		w3(y)

T_1	T_2	T_3
r1(x) r1(z) w1(x)		
	r2(z)	
		r3(x)
	r2(y)	
		r3(y)
	w2(z)	
		w3(y)
	w2(y)	

T_1	T_2	T_3
r1(x) r1(z) w1(x)		
	r2(z) r2(y) w2(z)	
		r3(x) r3(y) w3(y)
	w2(y)	

Fig. 8.22: Schedule

T3 writes y which is again written by T2, so we cannot swap w3 (y) and w2 (y). Hence schedule 52 is not a serializable schedule. Serializable graph is

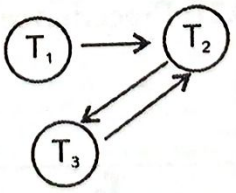


Fig. 8.23: Precedence Graph

Example 3. Consider the schedule S1. Determine whether each schedule is strict, cascadeless, recoverable or non-recoverable. Determine the strict recoverability condition that each schedule satisfies.

S1: r1 (x); r2 (z); r1 (z); r3 (x); r3 (y); w1 (x); C1 ; w3 (y); C3 ; r2 (y); w2 (z) ; w2 (y) ; C2 ;

Solution. Schedule S1 is

T_1	T_2	T_3
r1 (x)		
	r2(z)	
r1(z)		
		r3(x)
		r3(y)
w1(x) Commit		
		w3(y) Commit
	r2(y) w2(z) w2(y) Commit	

Fig. 8.24: Schedule

Condition of Strict Schedule

A transaction can neither read nor write an item x until the last transaction that wrote x has committed or aborted.

Schedule S1 is not strict because T3 reads x before T1 writes to x. If r3 (x) operation would have been after T1 commits then S1 would have been a strict schedule.

Condition for Cascadeless Schedule

Every transaction in a schedule reads only item that were written by committed transaction. Schedule S1 is not cascadeless because T3 reads a data item X before T1 writes to X and before T1 commits. Hence, T3 does not read data item from committed transaction.

(GGSIPU 2011)

Condition for Recoverable Schedule

If T_j has read any data item written by T_i, then T_j must commit after T_i. To check whether the schedule is recoverable we need to abort each transaction schedule once.

If T1 aborts → T3 commits → T2 commits, then schedule S1 is recoverable because T1 write X and no other transaction then reads or write X. This means rolling back of T1 does not affect T2 or T3.

If T1 commits → T3 aborts → T2 commits then schedule S1 is not recoverable because T2 reads data item Y which T3 had written earlier. Since T3 is rolled backed, T2 reads a value which does not exist.

If T1 commits → T3 commits → T3 aborts then, schedule S1 is recoverable because any operation of T2 does not affect T1 or T3. Now, the strictest condition of schedule S1 is T3 commit → T2 commits.

Example 4. Consider the precedence graphs shown below. Is the corresponding schedule conflict serializable? Explain.

(a)

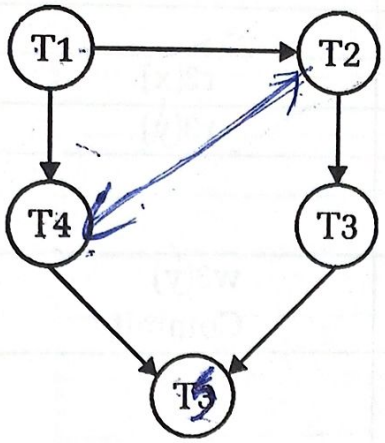
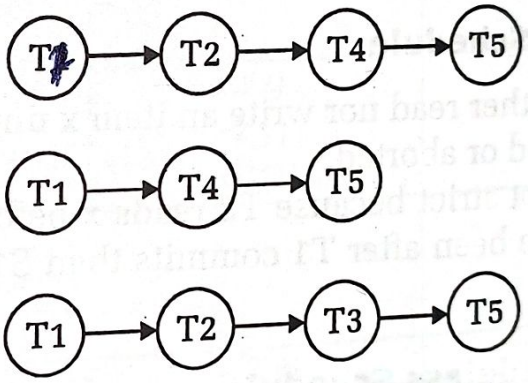
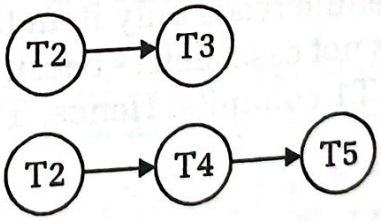


Fig. 8.25: Precedence Graph

Solution. Edge Starting from T1



Edge starting from T2



Edges starting from T3

Edges starting from T4

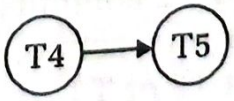
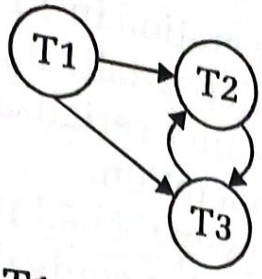


Fig. 8.26: Precedence Graph

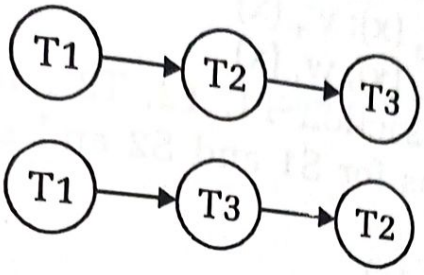
Edges starting from T5
None

Since in above graph we did not find any cycle so the graph belongs to soe conflict serializable schedule.

(b)



Solution. Edges starting from T1



Edges starting from T2

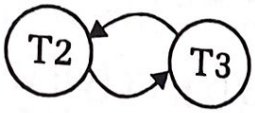


Fig. 8.27: Precedence Graph

Since a cycle exist, so the graph does not belong to a conflict serializable schedule.